Naval Research Laboratory

Washington, DC 20375-5320

NRL/MR/5540--97-7991

Design and Assurance Strategy for the NRL Pump

Myong H. Kang
Andrew P. Moore
Ira S. Moskowitz

Center for High Assurance Computer Systems
Information Technology Division

December 31, 1997

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE December 31, 1997	3. REPORT TYPE AND DATES COVERED	
4. TITLE AND SUBTITLE Design and Assurance Strategy for the NRL Pump			5. FUNDING NUMBERS PE - 61153N15
6. AUTHOR(S) Myong H. Kang, Andrew P. Moore, and Ira S. Moskowitz			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Research Laboratory Washington, DC 20375-5320			8. PERFORMING ORGANIZATION REPORT NUMBER NRL/MR/5540--97-7991
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Office of Naval Research Arlington, VA 22217-5660			10. SPONSORING/MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES			
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.			12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Developing a trustworthy system is difficult because the developer must construct a persuasive argument that the system conforms to its critical requirements. This assurance argument, as well as the software and hardware, must be evaluated by an independent certification team. In this paper, we present the external requirements and logical design of a specific trusted device, the NRL Pump, and describe our plan, called the assurance strategy, to create the eventual assurance argument. Our assurance strategy exploits currently available graphical specification, simulation, formal proof, and testing coverage analysis tools. Portions of the design are represented by figures generated by the Statement tool-set, and we discuss how those tools, and covert channel analysis, will be used to show that the logical design conforms to its external requirements. We conclude with some remarks on a possible physical architecture.

14. SUBJECT TERMS Computer device			15. NUMBER OF PAGES 25
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL

CONTENTS

Design and Assurance Strategy for the NRL Pump

Myong H. Kang, Andrew P. Moore, and Ira S. Moskowitz[1]

Center for High Assurance Computer Systems

Information Technology Division, Mail Code 5540

Naval Research Laboratory

Washington, DC 20375

Abstract

Developing a trustworthy system is difficult because the developer must construct a persuasive argument that the system conforms to its critical requirements. This *assurance argument*, as well as the software and hardware, must be evaluated by an independent certification team. In this paper, we present the external requirements and logical design of a specific trusted device, the NRL Pump, and describe our plan, called the *assurance strategy*, to create the eventual assurance argument. Our assurance strategy exploits currently available graphical specification, simulation, formal proof, and testing coverage analysis tools. Portions of the design are represented by figures generated by the Statemate toolset, and we discuss how those tools, and covert channel analysis, will be used to show that the logical design conforms to its external requirements. We conclude with some remarks on a possible physical architecture.

1. Introduction

In the last few years, the landscape of multilevel secure (MLS) computing has changed dramatically. Researchers and practitioners in the information security arena realize how difficult it is to build general purpose high-assurance MLS computers and software. Over the past 20 years, only a handful of high-assurance MLS computers has been built. Those high-assurance computers are rarely used in operational environments because

- they are relatively expensive,
- they lack user friendly features and development environments,
- they lag current commercial systems because of the time required for evaluation and certification, and
- they do not provide scalable solutions for secure distributed computing.

Despite the lack of satisfactory solutions, information security has become a more important issue because of the trend to open and distributed computing, which increases the vulnerability of the system to attack. Hence, it is increasingly important to develop scalable security solutions that do not depend on general purpose MLS systems. Such

[1]Research supported by the Office of Naval Research
Manuscript approved June 27, 1997

security solutions should use commercial products for general purpose computing and special purpose trusted devices for the separation of data at different security levels. This paper describes the software design and the assurance strategy of a high-assurance security device, the NRL Pump, which is one of the security components for a proposed security architecture [KFM].

2. NRL Pump Overview

In 1993, Kang and Moskowitz introduced the Basic NRL Pump [KM93]. Further results [KML] extended the Basic NRL Pump to the network environment -- the Network NRL Pump, which is the focus of this paper. For brevity, we refer to the Network NRL Pump simply as "the Pump" in this paper.

Suppose messages are sent from an enclave operating at a low security level (Low) to an enclave operating at a high security level (High). The security requirement is that information may be sent from Low to High, but not the reverse. Although this requirement seems simple, it is often quite difficult to satisfy. Applications sending messages from Low to High require an acknowledgement (ACK) that the message was successfully transmitted. This ACK is required both for reliability and recoverability. One might think that with the ACK as benign as possible, basically just `last message successfully received`, there would be no security problem since we are not allowing any information to be padded into the ACK. However, if the timing of the ACK to Low is under the control of High we have the possibility of a covert communication channel [L]. To be precise, we have a potential timing channel [L][W][MM92][MM94]. That is, High can send information, in the sense of Shannon [S], to Low by varying the ACK arrival times to Low, after Low has sent a message to High.

Perhaps this channel seems a minor vulnerability, but related channels have been demonstrated to have significant capacities in real systems [G], and if all other means of communication are cut-off then an exploiter may use the only means possible -- the timing of the ACKs. The potential damage caused by timing channels such as this has been well studied in [KM95]. The Pump limits this timing channel without constraining throughput. It also enforces a fairness criterion among the different users.

The Pump places a non-volatile buffer between Low and High. A system in Low sends a message to the Pump. The Pump stores this message in this buffer and sends an ACK to the sender. The timing of the ACK is stochastically modulated based upon a moving average of past High activity. The Pump asynchronously forwards the message to High. Details of the Pump algorithms are documented elsewhere [KML].

The Pump is configured as a single hardware box that has interfaces to a High LAN, a Low LAN, and an Administrator Workstation (which is used to load configuration information into the Pump and to monitor its operation as necessary). The Pump supports a specialized protocol (the Pump Protocol) across the LAN interfaces.

The ability to support a variety of applications is provided by software called *wrappers*, which runs on the application systems in the Low and High enclaves, that communicate with the Pump over their respective LANs. Each wrapper is further divided into an application-dependent part, which can be tailored to support the particular set of objects or calls the application it expects to see, and a Pump-dependent part, which is a library of routines that implement the Pump protocol. These functions can be called as required by the application-dependent routines. Note that only application programs that can operate with very little information returned to the sender from the receiver (e.g., applications that use asynchronous communication) can use the Pump, since the reverse path is limited to simple ACKs.

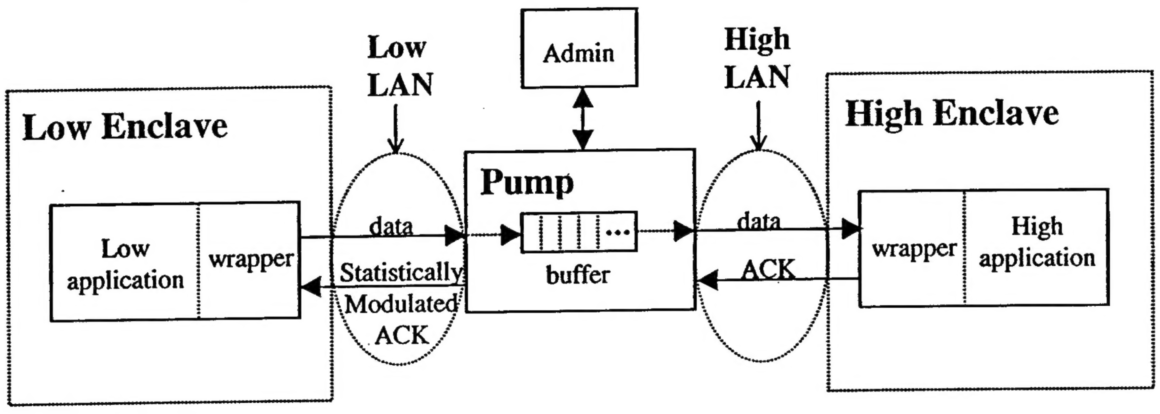


Figure 1: The Pump, wrappers, and applications

Figure 1 shows the Pump with its administrator workstation, the Low and High LANs, and the wrappers and applications with which the Pump communicates. Confidentiality properties of the Pump depend solely on itself and not on the wrappers. Wrapper software, both application-dependent and Pump-dependent parts, is not security critical and can be altered or replaced without affecting system security.

Each wrapper consists of two parts: a Pump specific part and an application specific part. The Pump specific part supports Pump Application Programming Interface (API) on one side and the Pump protocol on the other side. The application specific part of the wrapper provides application specific protocol on one side and Pump API on the other side as shown in Figure 2.

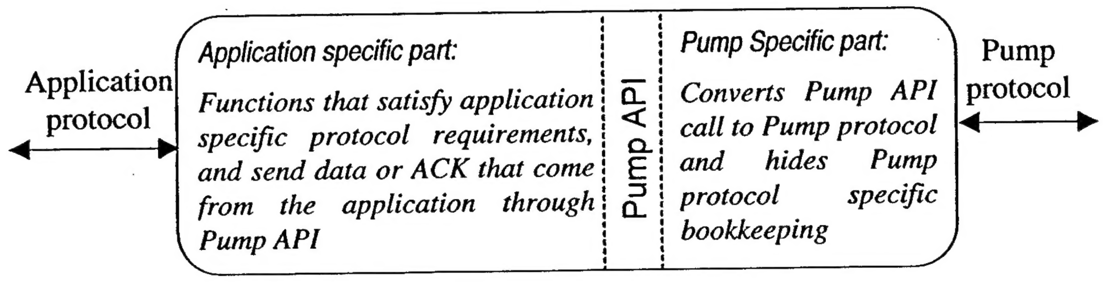

Application protocol
Application specific part:
Functions that satisfy application specific protocol requirements, and send data or ACK that come from the application through Pump API
Pump API
Pump Specific part:
Converts Pump API call to Pump protocol and hides Pump protocol specific bookkeeping
Pump protocol

Figure 2: The Structure of wrappers

Some examples of Pump API function calls are `connectToPump(highAddress)`, `sendData(message)`, `sendACK(id)`,etc.

Note that the low wrapper is a *proxy* of the high application program that (1) receives messages from the low application program and delivers them to the Pump, and (2) receives ACKs from the Pump and generates application specific ACKs. Also note that sometimes one application message from a low application may be transformed into several Pump messages. Similarly, the high wrapper is a *proxy* of the low application program that (1) receives a message from the Pump and delivers it to the high application program and (2) receives application specific ACKs and converts them to Pump specific ACKs.

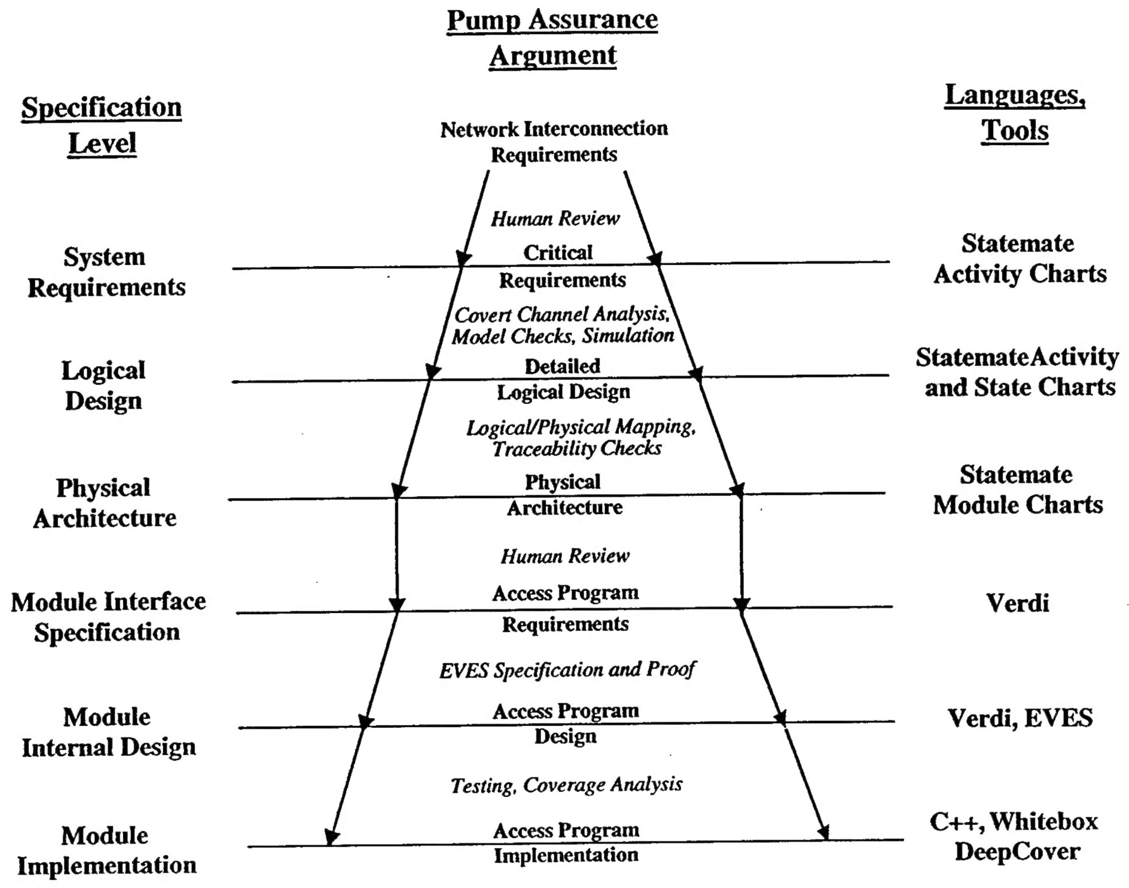
Our current interest is to demonstrate the use of the Pump in a SINTRA [FKMCL] database. Low is one database and High is another database; data in Low is to be replicated to High. The databases are Sybase databases and a Sybase replication server is used in conjunction with the Pump to forward data from Low up to High.

3. Assurance strategy

Information systems that effectively and inexpensively counter security threats isolate the security-critical function of the system architecture in simple, well-defined and reusable components. A detailed explanation, called the assurance argument, describes why this isolation is effective and why the critical components are *trustworthy.* The critical components are *trusted* to correctly carry out the security-critical function. The development of a trustworthy system is not easy because an assurance argument must be constructed by the developer and evaluated by an independent certification team. The argument must instill high confidence that the system does what it is supposed to do, and only what it is supposed to do. Constructing a convincing assurance argument requires a comprehensible development process and an implementation that clearly conforms to its critical requirements.

Judicious use of formal methods can strengthen a system's assurance argument, because the tools of mathematics and logic can be applied to assure that critical properties hold. Many evaluation criteria for trusted computer products and systems reflect this fact [T] [C] [I]. However, increasing the formality of an argument does not necessarily make it more convincing to an independent certifier unfamiliar with these tools. Constructing a persuasive and cost-effective argument often requires the use of many different languages, methods and tools --- both formal and informal. Formal specifications and analyses must be intuitively presented in the context of the overall assurance argument or much of their power to persuade may be lost [MP].

The process that we have adopted for developing the Pump integrates the specifications and analyses with structured system documentation. This process clarifies the relationship between the refinement of Pump functionality and the argument that the Pump satisfies its critical requirements. Figure 3 illustrates our process for constructing the Pump's assurance argument.



Pump Assurance Argument
Specification Level
Languages, Tools
Network Interconnection Requirements
Human Review
System Requirements
Critical Requirements
Statemate Activity Charts
Covert Channel Analysis, Model Checks, Simulation
Logical Design
Detailed Logical Design
StatemateActivity and State Charts
Logical/Physical Mapping, Traceability Checks
Physical Architecture
Physical Architecture
Statemate Module Charts
Human Review
Module Interface Specification
Access Program Requirements
Verdi
EVES Specification and Proof
Module Internal Design
Access Program Design
Verdi, EVES
Testing, Coverage Analysis
Module Implementation
Access Program Implementation
C++, Whitebox DeepCover

Figure 3: Pump assurance strategy overview

The primary levels of system refinement and documentation are shown along the left side of Figure 3. Along the right side are the specification languages and tools that contribute to the implementation, analysis and verification of the Pump, e.g., I-Logix Statemate^{TM} graphical specification and simulation tools [HLNPPSST], ORA Canada's Verdi/EVES formal verification environment [CKMPS][KPSCM] and Reliable Software Technology's Whitebox DeepCover^{TM} testing coverage analysis tools [RST]. The result of integrating the use of the languages and tools on the right into the levels of system documentation on the left will be the Pump's assurance argument, which corresponds to the center of Figure 3 (the area between the arrows). Slanted arrows indicate a refinement of a specification to a more detailed specification or implementation; vertical arrows indicate a translation of a specification from one semantic domain to another at a comparable abstraction level. The increase in width of the argument from top to bottom reflects additional detail specified and reflected in the assurance argument at the lower levels.

A variety of formal and informal techniques allow reasoning across five semantic domains: English narrative, Statemate logical design, Statemate physical design, formal

Verdi PDL specification, and C++ code. The network interconnection requirements are expressed in English. The Pump's critical requirements are also expressed in English but in terms of the primitives of a functional view of the Pump specified graphically in Statemate activity charts. This logical view of the Pump is refined using a combination of Statemate activity and state charts, which details the behavioral view of the Pump. The activities and behavior of the logical design are mapped to a physical architecture described in terms of Statemate module charts. This physical architecture is mapped to a Verdi specification of the interface function (i.e., access program) requirements of each module. This specification provides the "oracle" to which the implementation must be shown to conform. Verification proceeds either through formal proof using the EVES verification system or by thorough testing using the Whitebox DeepCover tool for coverage analysis. The type of verification performed depends on the complexity and type of the requirement, e.g., functional, security, performance, etc., and the complexity of the code. The theory of software testability [VM] may be used to determine the likelihood that testing will uncover flaws in the implementation.

We have studied the behavior and vulnerability of the Pump algorithms during normal operation [KML], but not during connection initialization, error handling, or error recovery. Implementation may introduce vulnerabilities such as the overuse of Pump's resources due to the high consumption by a single connection or the revelation of the number of active connections. We are developing a fully functional Pump prototype that reflects the logical design of the Pump to study these vulnerabilities within the operational environment. Functions that need to be clarified include:

- administrative requirements of the Pump, such as initialization of the Pump, connection establishment procedure, and monitoring the activities of the Pump; and
- error handling and audit requirements.

The Pump's assurance strategy must yield easy-to-understand mappings from the critical system requirements to the design. To support these mappings, the Pump design is based on the following principles:

1. Define clearly the task of each module (i.e., thread or object in the logical design prototype),
2. Separate modules that interact with Low from modules that interact with High, called low and high modules respectively, and
3. Reduce as much as possible communication between low modules and high modules so that the security (especially confidentiality) can be easily monitored and verified.

4. Notation

Sections 5 and 6 of this paper present an overview of the system requirements and logical design, which are the first two levels of specification of the Pump shown in Figure 3. Section 7 describes the analyses performed on these specifications including a covert channel analysis, a Statemate analysis of their logical consistency and completeness, and Statemate simulations. The last section discusses future work involving the refinement of the logical design to the physical implementation of the hardware Pump, as will be carried out in the last four levels of refinement in Figure 3.

The language of Statemate forms the basis for the specifications that follow. Table 1 presents the primary graphical notation used in this paper. Two types of Statemate graphical charts are used: activity charts, which represent a functional view akin to data flow diagrams, and state charts, which represent a behavioral view akin to state machine diagrams. State charts describe the behavior and control of activities in an activity chart; thus, an activity chart may be associated with a controlling state chart as shown in the table. Statemate distinguishes two types of flows between activities in an activity chart: flows of data items, which are represented by solid arrows, and flows of control (events or conditions), which are represented by dashed arrows. An arrow between states in a state chart must be labeled with a trigger that causes the state transition. These triggers have the general form E[C]/A, where E is an event (an observation of the system that occurs instantaneously), C is a condition (an observation that is either true or false), and A is an action (which may cause other events to trigger or conditions to change). For example, in Figure 9, E/A is `CONNECTION_RQSTD/get!(CNCT_RQST,LO_RQST)` where the `get!` call receives `LO_RQST` message from `CNCT_RQST` queue. Statemate also has various connectors that permit graphically decomposing triggers into a series of branches. These connectors are used solely to clarify the specification by reducing the number and length of arrows between states.

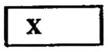
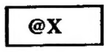
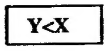
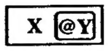
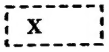
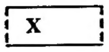
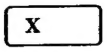
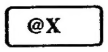
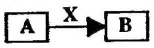
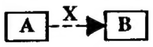
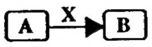
	- an activity named X
	- an activity named X that is refined in a lower level chart also named X
	- an instance Y of a generic (parameterized) activity X
	- an activity X with controlling state chart Y
	- an activity X that is external to the chart being elaborated
	- a place to store data item X
	- a state named X (Note rounded edges)
	- a state X that is refined in a lower level state chart also named X
	- a flow of data item X from activity (or data store) A to activity (or data store) B
	- a flow of control element X from activity A to activity B
	- a trigger X that causes transition from state A to state B

Table 1: Statemate Graphical Notation

5. System Requirements --- External View of the Pump

In this section, we treat the Pump as a black box and describe its requirements from an outsider's point of view. The Statemate activity chart showing the data flows from this viewpoint is shown in Figure 4.

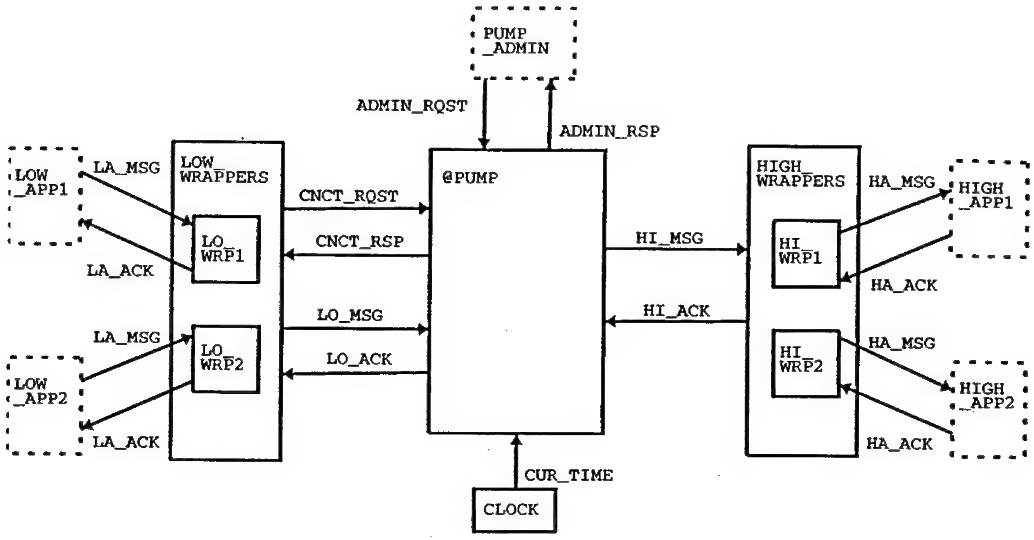


Figure 4: External View of the Pump

System Configuration and Administration

The Pump can be thought of as simple network router that connects a low network to a high network. However, we do not want the Pump to act as a general-purpose router that can accept a message from a lower security level process and route that message to any high security level process. Allowing such uncontrolled behavior can cause security and availability problems. For example, any low process could establish a connection to any high process and thus waste Pump resources. In addition, a low Trojan Horse process could “ping” high (Trojan Horse) processes to see if such processes exist or not. The Pump should prevent such arbitrary exploitation of the Pump for security and availability reasons, but at the same time should provide flexible services to many applications for usability.

To avoid these problems, we require processes that use the Pump to register their addresses with the Pump administrator. The Pump administrator, who will verify the legitimacy of the registration, can enter the addresses of registered processes into the Pump's configuration file. This configuration file can be changed and reloaded anytime during the normal operation of the Pump.

Note that, in general, the Pump communicates with COTS applications only via wrappers that understand both the Pump and application protocols. Hence, the Pump's registered processes are most likely the low and high wrappers. When the wrapper is registered, it also identifies the type of application (either recoverable or non-recoverable, see below) with which it interfaces.

The Pump has a configuration file that contains:

1. Pump initialization information (e.g., window size, and maximum number of connections)

2. Registered low and high processes, and the type of applications that the process interacts with (i.e., either recoverable or non-recoverable), and
3. A set of allowable connections. This information is used for network access control when a low process sends a connection request to the Pump.

A Pump administrator, who is cleared for high data, manages this file. Since the Pump checks the configuration file only when a connection is established, this file can be reloaded dynamically.

The Pump has an external administrative interface for loading configuration files, requesting the status of the Pump, etc. When the Pump administrator requests the status of the Pump, it returns the status of active and aborted connections (e.g., how long the connection was active, idle time). When the idle time of a connection is too long, the Pump administrator has the option to "kill" the connection.

The Pump also maintains a well-known port to which a low process can send a connection request to a specific high process. This is shown as the `CNCT_RQST` and `CNCT_RSP` flows in Figure 4.

Recovery

The Pump must provide a recoverable service. That is, once a Low wrapper receives an ACK from the Pump for a given message, it must be able to safely assume that the message will be delivered to the corresponding High wrapper by the Pump, even if power failures or system crashes occur, either in the Pump or the High Wrapper.

Not all applications require the same kind of recoverable service, however, and the Pump's external interface permits applications to request a recoverable or non-recoverable connection; this choice determines how the Pump behaves if the connection is aborted. For example, suppose the Pump delivers messages between an FTP client and server. FTP is not a recoverable application; so, if there is an abnormal disconnection in the middle of a file transfer, the FTP client and server do not expect any recovery when they resume the connection. However, if a connection is abnormally broken between a Sybase replication server and SQL server, they do expect recovery after the connection is resumed because they are recoverable applications.

Different recoverable applications have different recovery procedures, so their wrappers must maintain the necessary information for recovery. In the case of a Sybase replication server and SQL server, they exchange the last message that the replication server sends to the SQL server for synchronization. Hence, the wrapper of the replication server has to keep the last message. Since the wrapper cannot predict when the connection will be aborted, it has to write every message to persistent storage. In general, the low and high systems in which the wrappers reside are not recoverable systems. Maintaining persistent messages in a non-recoverable system is usually a very expensive operation (e.g., the need to write every message to disk and synchronize).

The Pump is a recoverable device, hence, maintaining an extra persistent message for recoverable connection is not as expensive as maintaining a persistent message in a non-recoverable system. Therefore, the Pump is designed to maintain the last message it receives from Low if the connection is recoverable and the connection is abnormally disconnected, even if all other messages are already delivered to High. However, the Pump cannot keep the last message forever. Hence, the Pump will maintain the last message from the aborted recoverable connection only for *T* (a configuration parameter) hours. The administrator of the Pump can always reclaim the resources from the aborted connection after *T* hours. Upon receiving the command from the administrator to reclaim the resources for the, aborted connection, the Pump tries the last effort to "flush" the undelivered message to High and then claims the resources.

The normal recovery procedure for an aborted recoverable connection is as follows. After Low and High re-establish the connection, the Pump delivers all undelivered messages from the previous session to High. The Pump then sends the last message that it delivered to High (which is exactly the last message it received from Low) back to Low for synchronization purposes (unless that message was the connection close request, in which case nothing is delivered to Low). After this synchronization, Low sends new data messages to the Pump. Of course, there may be cases where the last messages do not provide sufficient information for synchronization. For such applications, the wrappers have to maintain extra information for synchronization.

Message classes and connection establishment procedure

For recoverability reasons, the Pump is designed to operate at the application layer [KMMP]. Hence, it communicates to Low and High through its own protocol (i.e., Pump message). There are two classes of Pump messages: data messages and control messages. The inheritance structure of message classes in OMT notation[R] is shown in Figure 5.

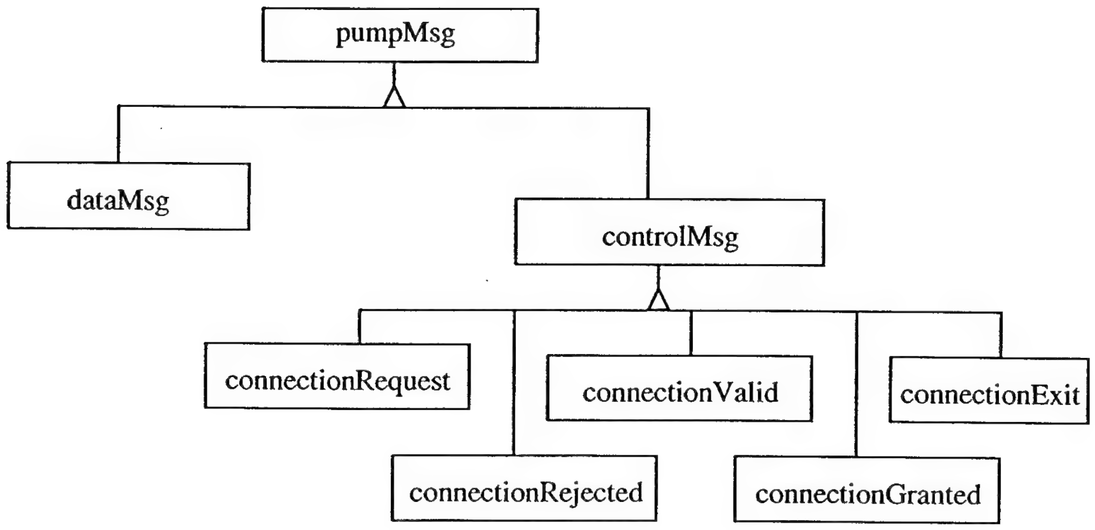


Figure 5: Message inheritance structure in OMT notation

Every Pump message has 7 bytes of header field and an arbitrary length of data field, as shown in Figure 6.

2 bytes (data length)	1 byte (type: either data or control)	4 bytes (extra header)	arbitrary length data field (data length field in the header specifies its length)

Figure 6: Pump message structure

The first two bytes specify the length of the data field, and the next byte tells the type of message: data or control. The next 4 bytes of the header will have different meanings for different types of messages, such as connection ID and message ID for data messages, and the type of control message and version number for control messages. Control messages are exchanged mainly to set up Low to High connections through the Pump. Data messages are used to send data from Low to High.

When Low sends a connection request message (`connectionRequest`) to the Pump, it identifies itself with its own address and the type of application (i.e., recoverable application or non-recoverable application). It also specifies the high address that it wishes to connect to. The Pump will check the configuration file to determine if the request is permitted. If the Low and High addresses match with the connectivity table in the configuration file, the Pump will send the connection valid message (i.e., `connectionValid`) to Low. If the connection request was originated from an unregistered low process then the request is ignored. If a registered Low process requests a connection that is not specified in the configuration file (i.e., illegal connection request), a connection rejected message (`connectionRejected`) will be sent to Low. When Low receives a `connectionValid` message, it disconnects the current connection and is ready to accept a new connection from the Pump. This redundant connection procedure is intended to verify Low's address[2]. Low will use this new connection to transmit data.

Registered high processes are always ready to accept a connection from the Pump. Once the Pump validates the connection request from Low, it initiates a new connection to High by relaying the `connectionRequest` message that came from Low to High. High validates the request and sends a `connectionValid` or `connectionRejected` message to the Pump. When the new connection is established, the Pump sends a `connectionGrant` message with initialization parameters to High. If the connection is recoverable and the previous connection is abnormally disconnected then the Pump will send undelivered messages from the previous session to High. If the connection to High is successfully established and all undelivered messages are cleared then the Pump establishes a connection to Low and

[2] There is always a danger that a "bad process" can send a connection request by pretending that it is some other process. To prevent such attacks, a strong authentication mechanism (e.g., digital signature) could be used. The current implementation of the Pump does not use a strong authentication mechanism because: (1) using digital signatures for every message would be too costly in terms of performance, (2) audit can detect suspicious activities (e.g., a legitimate connection request is refused due to the active connection between the same Low and High, suspiciously long connection time), and (3) such mechanisms can be provided external to the Pump.

sends `connectionGrant` message to Low. If the connection is recoverable and the previous connection is abnormally disconnected then the Pump will send the last data message it received from Low for synchronization purposes. If the last message is the connection close request, then the Pump does not send any message to Low for synchronization purposes. If the connection to High cannot be established or all undelivered messages from the previous session cannot be cleared then the Pump establishes a connection to Low and sends `connectionExit` message to Low. `connectionExit` messages are sent to Low and High when the Pump is ready to shut down the connection due to any error or administrator's request.

Once the Low to High connection is established, the data message is used to send information from Low to High. ACK is a special data message that has zero data length (i.e., the first two bytes of the message are zero) that can be sent from High to the Pump, and from the Pump to Low.

There is another special data message that requests normal "connection close" from Low to the Pump, and from the Pump to High. This message is used at the end of normal data transmission. Logically it should be a control message. However, this special message has to propagate from Low to High through the Pump in sequence (i.e., all true data messages have to be delivered to High before this message is delivered to High). When designing the Pump, one does not want to introduce an extra communication channel from Low to High for this connection close message. By sending the connection close message as a data message through the established connection, we not only avoid the need for an out-of-band signal, we can assure that it will be processed in the correct order (i.e., by the time High receives this message, all other data messages should be processed). Hence, the data type message from Low (Pump) with data length zero is interpreted as connection close request by the Pump (High). Note that, in general, the connection close message is create by low wrappers not by low applications.

6. Logical Design --- Internal Structure of the Pump

In this section, we describe the internal structure of the Pump. The Pump has three types of threads: the main thread (MT), trusted low threads (TLT), and the trusted high threads (THT). The Pump also has three types of data structures: connection table (one per Pump), connection buffer (one per active or aborted connection), and Pump messages, which were introduced in section 5. The rest of the structures are introduced in this section. We especially emphasize the mapping of the function from the system requirements to the threads and objects of the logical design. The high-level structure of the Pump is shown in Figure 7.

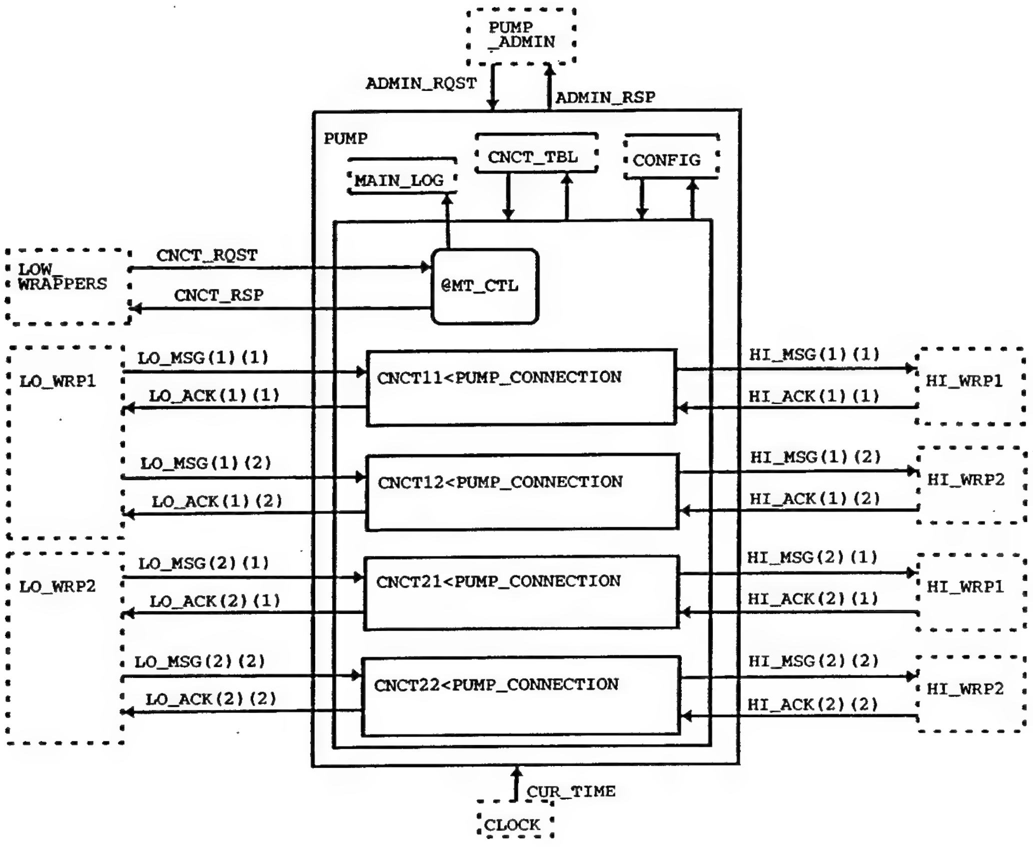


Figure 7: Internal structure of the Pump

Connection Table

The Pump maintains a connection table that records the status of all active and aborted connections. If there is a legal connection in the configuration file, there is a maximum of one entry in the connection table (i.e., if the connection is neither active nor aborted then there is no entry in the connection table). Each entry in the connection table records the status of the connection (active or aborted), the address of its connection buffer, the addresses of High and Low, pointers to THT and TLT (null if the connection is inactive), and the time of the last activities of either THT or TLT.

Connection buffer

Each connection between a low sender and a high receiver has one FIFO bounded buffer controlled by a monitor and accessed by two threads: TLT, which puts messages in the buffer and THT, which removes them. The connection buffer stores an array of handles of data messages and a variable that records the moving average of the outgoing message rate (THT's consumption rate) used by TLT to control the stochastic delay for ACKs to Low.

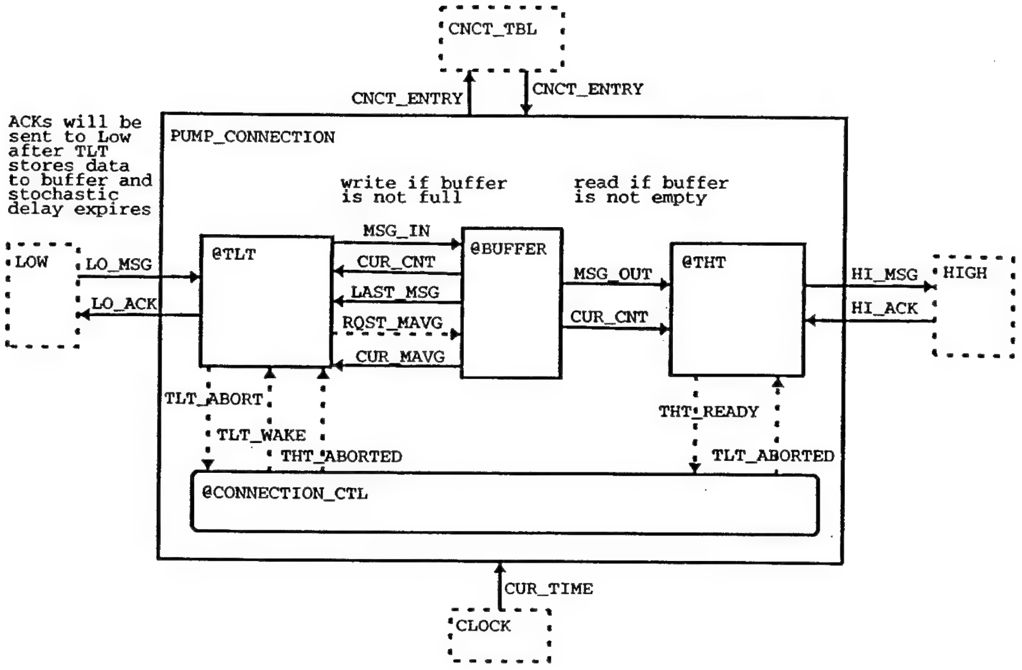


Figure 8: A Pump Connection

A connection buffer is created when a new, valid connection is requested from a Low to a High, if there is no pre-existing connection buffer from an aborted connection between the same pair of Low and High ports. A connection buffer is deleted when a connection terminates normally (with a connection close message).

Main thread (MT)

The role of MT is to initialize the Pump, which includes reading the configuration file, and to keep track of relevant information for each connection. MT also listens to the well-known port of the Pump to which Low sends `connectionRequest` messages as described in Section 5. In response to a valid request, MT first spawns a connection that consists of THT, TLT, and connection buffer. MT then sends a connection valid message (`connectionValid`) to Low. In response to an invalid request, `connectionRejected` will be sent to Low. Note that if the connection request is from an unregistered low process then MT ignores the request. The rest of the connection set up procedures are performed by both THT and TLT. After exchanging necessary control messages, Low starts sending data messages. Another responsibility of MT is to populate the connection table as it spawns a connection (e.g., status of the connection, pointers to THT, TLT, and connection buffer). The essential behavior of MT is shown in Figure 9.

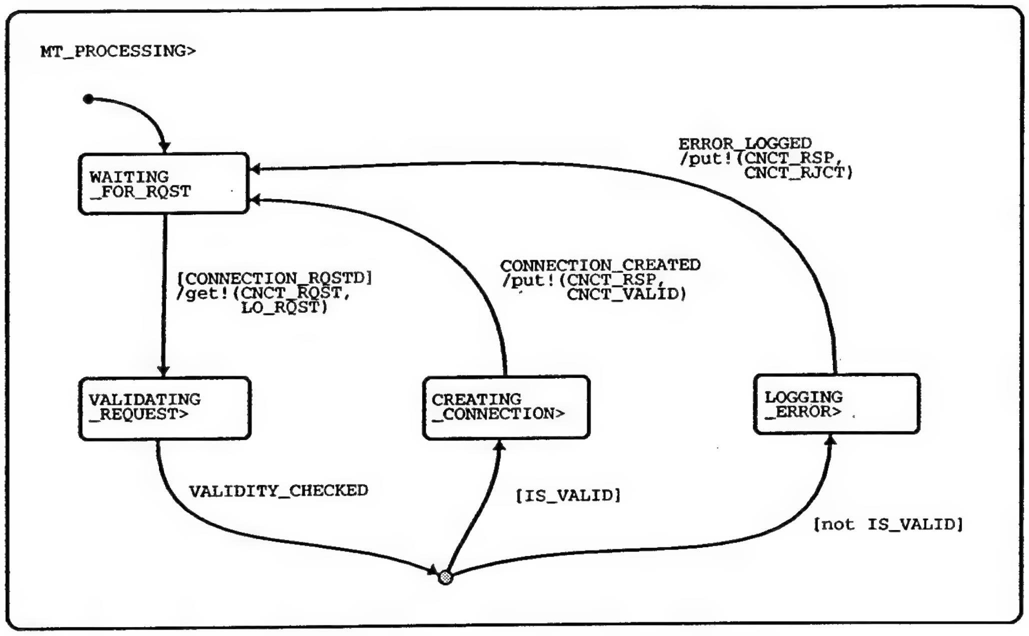

MT_PROCESSING>
WAITING _FOR_RQST
ERROR_LOGGED /put!(CNCT_RSP, CNCT_RJCT)
[CONNECTION_RQSTD] /get!(CNCT_RQST, LO_RQST)
CONNECTION_CREATED /put!(CNCT_RSP, CNCT_VALID)
VALIDATING _REQUEST>
CREATING _CONNECTION>
LOGGING _ERROR>
VALIDITY_CHECKED
[IS_VALID]
[not IS_VALID]

Figure 9: Behavior of main thread (MT)

Trusted high thread (THT)

When a new THT is spawned, it establishes a connection to High by sending `connectionRequest`. After it receives `connectionValid` message, THT then sends `connectionGrant` message that contains connection id, maximum message size, etc. THT then delivers any leftover data messages in the buffer from the previous (aborted) session to High. When the buffer is empty, it awakens TLT. THT keeps delivering messages as long as there are messages in the connection buffer. THT also updates the moving average based on ACK times from High. THT and TLT use a sliding window scheme with window size *w* that is specified in the configuration file (i.e., THT can send up to *w* data messages from the buffer without receiving ACKs from High). The Pump protocol requires High to send ACKs to THT in the same order it receives messages. If High violates the Pump protocol (e.g., by sending an out-of-sequence ACK), THT sends `connectionExit` message to High, disconnects it from High, and logs High's misbehavior. The essential behavior of THT is shown in Figure 10.

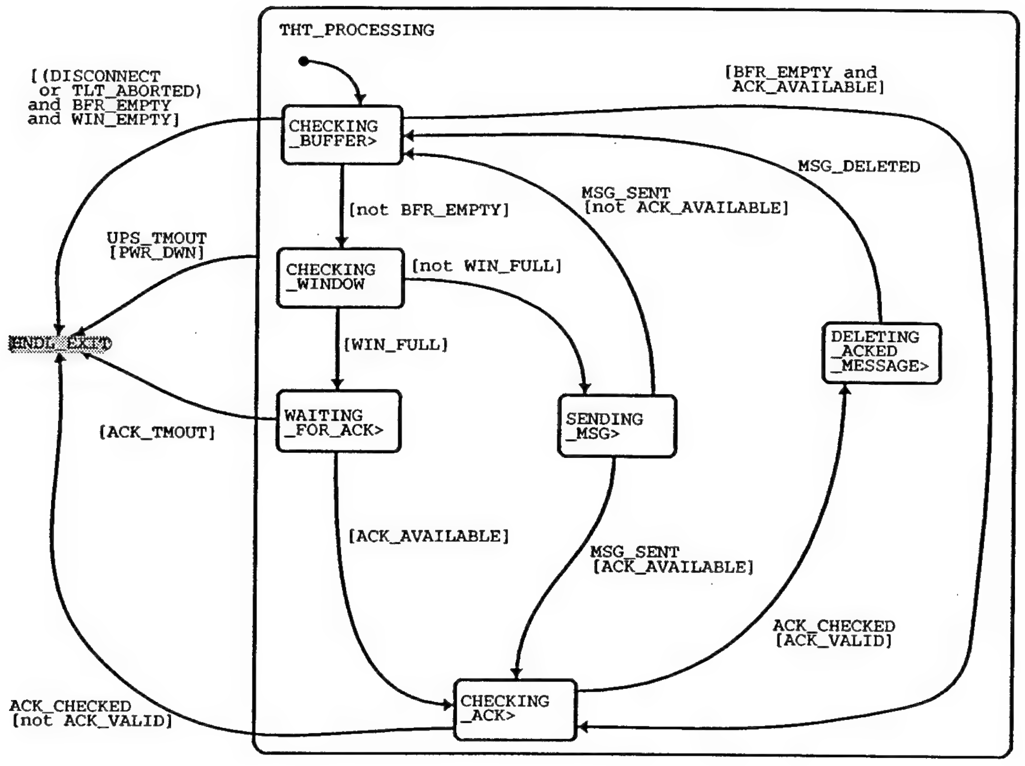


Figure 10: Behavior of trusted high thread (THT)

Trusted low thread (TLT)

When a TLT is created, it waits for THT to awaken it (i.e., wait until all undelivered messages from the previous session are delivered to High if the connection is a recoverable one). TLT then establishes a connection to Low and sends a `connectionGrant` message to Low. If the application is a recoverable one and the last message exists (which is not connection close message) that was received from the previous session (i.e., there was an abnormal disconnection in the previous session) then TLT also sends the last message it received from Low. TLT then starts to receive data messages from Low. Upon receiving a data message, it verifies message ID, connection ID, allocates memory, and then stores the handle of the message in the connection buffer. TLT also computes a stochastic random delay [KML] based on the moving average of THT's message consumption rate.

TLT receives up to *w* data messages without sending any ACKs to Low. TLT must acknowledge messages in the same order they are received from Low, despite the probabilistic delay. To maintain the order and the timing of the delayed ACKs, TLT maintains `ACKId` and `ACKQueue`. When TLT computes the delay, it stores the time value when the next ACK should be sent out in `ACKQueue`. All the time values in `ACKQueue` will be sorted in ascending order. As soon as the current time passes, the first time value in the queue, an ACK with `ACKId` should be sent out and `ACKId` should be incremented.

If the Pump has to close a connection abnormally (e.g., it receives a message with a wrong connection id), it will send a `connectionExit` message to Low. The essential behavior of TLT is shown in Figure 11.

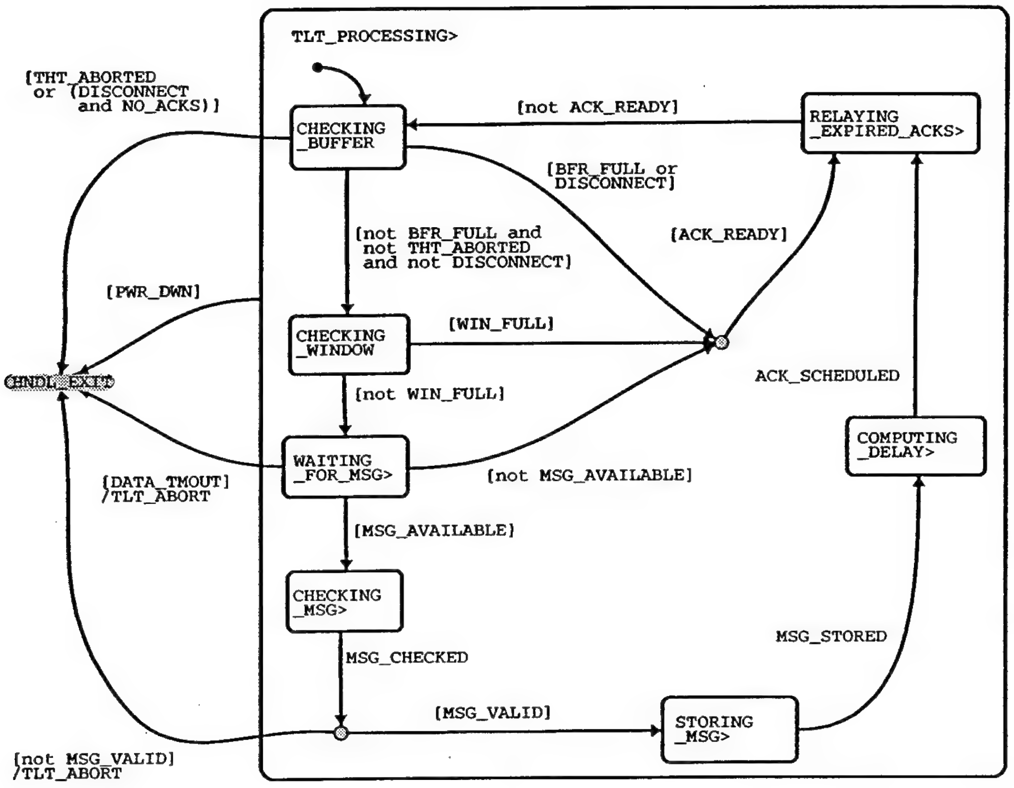


Figure 11: Behavior of trusted low thread (TLT)

Error/Failure Handling and Audit

Error handling is one of the most difficult parts of the design process because there is no theory or best way to handle errors. One important question that must be answered is "How smart should the Pump be for error recovery?" The smarter the Pump has to be, the more complex the software will be, and the harder it will be to assure that it behaves correctly. Hence, the Pump provides sufficient, but minimal, error recovery.

The Pump is designed to handle two specific failures: power failure and connection failure. The Pump's modules are designed to handle different types of failure independently and differently. We describe how each thread handles different types of failures. If more than one failure occur in series, the action for each failure will be activated in the order of failures. The error handling philosophy of the Pump is simple. *Avoid allocating resources for messages that cannot be delivered to High.* Hence,

1. If the source of the failure is the Pump then stop receiving messages from Low and try to deliver as many messages as possible to High.
2. If the source of the failure is the high side then stop receiving messages from Low immediately,
3. If the source of the failure is the low side then deliver as many already received messages as possible to High.

Power failure

The Pump has an uninterruptable power supply (UPS) that can send a power failure signal to the Pump. Each thread behaves as follows when it detects a power failure signal (and before each thread terminates itself):

- **TLT:** It immediately empties ACKQueue by sending all necessary ACKs to Low immediately (regardless of the delay calculations) and then sends `connectionExit` message.
- **THT:** It continually empties the connection buffer by delivering data messages to High for a certain fixed period of time (i.e., this fixed time depends on the ability of the UPS). When the connection buffer is emptied or the fixed time is over, it sends a `connectionExit` message to High. If the connection is a recoverable one, it marks the connection aborted and saves undelivered messages (i.e., connection buffer).
- **MT:** It immediately stops receiving `connectionRequest` from Low. It then waits until all THTs and TLTs terminate themselves. It then saves the connection table for the recovery.

Connection failure

Connection failure for the Pump can occur in two ways: Low-to-Pump connection (called Low connection) failure and Pump-to-High connection (called High connection) failure. Once a connection from Low to High is established, MT does not play any role for handling connection failure.

When a Low connection fails:

- **TLT:** When TLT detects the Low connection failure, it error logs the failure.
- **THT:** It continues to empty the connection buffer by delivering data messages to High. When the connection buffer is emptied, it checks whether TLT is terminated. As soon as it detects that TLT is terminated, it sends a `connectionExit` message to High. If the connection is a recoverable connection and the last message is not a connection close request, it marks that the connection as aborted and saves the connection buffer.

When a High connection fails:

- **THT:** When THT detects the High connection failure, it error logs the failure. If the connection is a recoverable one and the last message it sends was not the connection close request then it marks the connection as aborted.
- **TLT:** As soon as TLT detects that THT has died, it immediately empties ACKQueue by sending all necessary ACKs to Low and then sends a `connectionExit` message. It then saves the connection buffer if the connection is a recoverable one.

Audit

Audit processes should be flexible so that the Pump administrator can control the overhead associated with the audit. One possibility is to specify the level of audit in the Pump configuration file. Some items that need to be audited are:

- Errors: Any event that causes the Pump to send a `connectionExit` message (e.g., connection failure and protocol failure) should be audited.
- Some normal activities, such as connection/disconnection, should be audited.
- Some exceptional Pump internal status statistics (e.g., buffer full) should be audited.

7. Analysis of the Logical Design

The logical design summarized in this paper has been specified within the Statemate toolset and has passed the correctness/completeness checks that Statemate requires. Of course, this does not insure that the design conforms to the Pump's critical requirements. This requires human review, a detailed covert channel analysis, and simulation of the logical design. We are currently soliciting comments on our design and have just started detailed conformance testing using the Statemate simulator. The rest of this section discusses the security of the logical design, including its covert channels.

The Pump is a secure one-way communication device that minimizes any direct or indirect communication from High to Low. From the above description of the Pump, it is clear that only THT talks to High, and MT and TLT talk to Low. However, MT only talks to Low during connection set up. Once the connection is set up, only TLT talks to Low. Although THT and TLT are trusted software, it is desirable to reduce any interaction between THT and TLT for assurance reasons. If there is any communication, it needs to be monitored carefully.

In the design of the Pump, there is no direct communication between THT and TLT. There are only three indirect communication paths between THT and TLT through shared memory during normal operation (i.e., not during connection set up or exit).

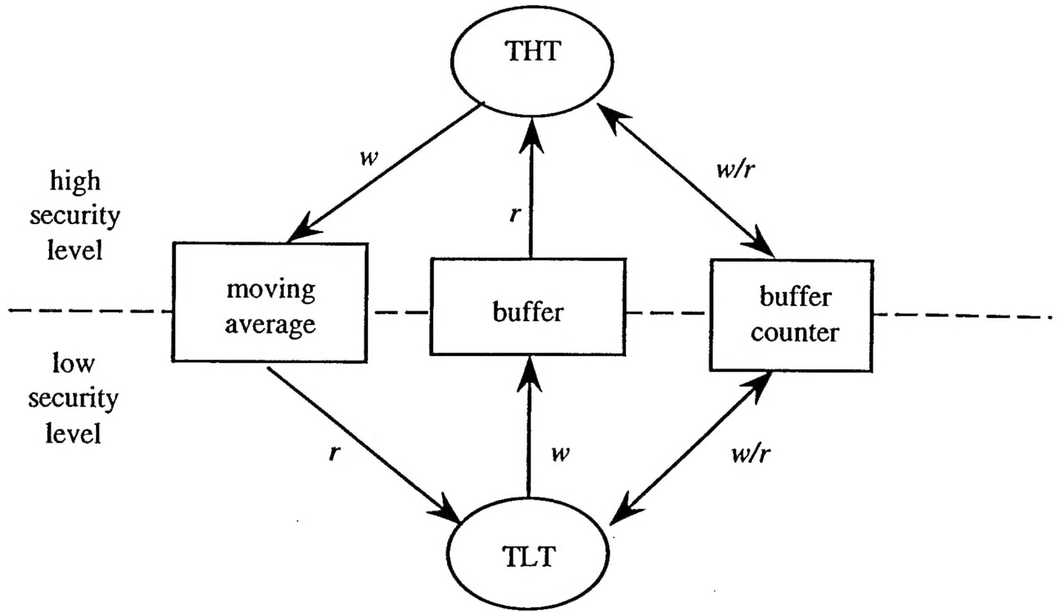


Figure 12: The pattern of interactions between THT and TLT

Only two of the three paths (see Figure 12) need to be monitored, because one path is a one-way upward path. The two other paths from THT to TLT and their impact upon Low have been extensively studied in the other Pump papers.

A major part of the security design for the Pump is its ability to mitigate covert timing channels from High to Low. However, some information can still be sent from High to Low because (1) Pump notifies Low when a connection is down and (2) recovery processes may be manipulated to leak some data. In designing a secure device that has any sort of realistic functionality it is impossible to eliminate all covert communication from High to Low (see the Small Message Criterion in [MK]). We minimize this additional covert channel by enforcing a minimum time τ between connection reestablishment and auditing of any connections that abort often. Hence, we have at worst introduced an additional covert channel with a capacity on the order of *(number of connections) bits per* τ . Furthermore, any covert channel that attempts to send meaningful amounts of information by using a disconnect/connect strategy will be easily detected by our audit process.

8. Summary and Future Work

In this paper, we have presented the software design and outlined the assurance strategy for the NRL Pump. We have focused on system requirements and logical design steps and described the mapping between them. The Pump software is structured so that it is easy to understand the mapping not only between system requirements and logical design, but also between logical design and physical architecture.

Our future work includes the design of a physical architecture. Since there are three distinct threads (i.e., MT, TLT, and THT) and objects that are shared among them, it may be reasonable to map those three threads into three processors. In that case

- one processor handles connection requests from Low and the communication to the administrator (i.e., MT is mapped to this processor),
- one processor handles all other communication to Low (i.e., TLT), and
- the last processor handles all communication to High (i.e., THT).

If only two processors are used then one processor can handle all communication to Low (e.g., connection requests and data from Low) and other processor can handle all communication to High and to the administrator.

References

[CKMPS] D. Craigen, S. Kromodimoeljo, I. Meisels, B. Pase, and M. Saaltink, "Reference Manual for the Language Verdi," Technical Report TR-91-5429-09a, ORA Canada, Ottawa, Ontario, September 1991.

[C] *Canadian Trusted Computer Product Evaluation Criteria.* Version 3.0e. January 1993.

[FKMCL] J. Froscher, M.H. Kang, J. MeDermott, O. Costich, and C. Landwehr, "A Practical Approach to High Assurance Multilevel Secure Computing Service," Proceedings of the Tenth Computer Security Applications Conference (1994).

[G] Girling, C. Gray. Covert Channles in LANs. *IEEE Trans. on Software Engineering, Vol. SE-13* (2) Feb., 1987.

[HLNPPSST] D. Harel, H Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. "Statemate: A Working Environment for the Development of Complex Reactive Systems," *IEEE Transactions on Software Engineering, 16*(4):403-414, April 1990.

[I] *Information Technology Security Evaluation Criteria.* Provisional Harmonised Criteria, ISBN 92-826-3004-8. Luxembourg, 1991.

[KFM] M.H. Kang, J. Froscher, and I.S. Moskowitz, "An Architecture for Multilevel Secure Interoperability," pp. 194-204, *Proc. 13th Annual Computer Security Applications Conference,* Dec. 1997, San Diego, CA, IEEE CS Press.

[KM93] M.H. Kang and I.S. Moskowitz, "A Pump for Rapid, Reliable, Secure Communication," *Proc. First ACM Conf. on Computer and Comm. Security* (1993).

[KM95] M.H. Kang and I.S. Moskowitz, "A Data Pump for Communication," NRL Memo Report 5540-95-7771, September 29, 1995.

[KML] M.H. Kang, I. S. Moskowitz, and D. Lee, "A Network Pump," *IEEE Trans. on Software Engineering, 22*(5: 329-338, (May, 1996).

[KMMP] M.H. Kang, I.S. Moskowitz, B.E. Montrose, J.J. Parsonese, "A Case Study of Two NRL Pump Prototypes," pp. 32-43, *Proc. 12th Annual Computer Security Applications Conference,* Dec. 1996, San Diego, CA, IEEE CS Press.

[KPSCM] S. Kromodimoeljo, B. Pase, M. Saaltink, D. Craigen, and I. Meisels, "EVES: An Overview. Technical Report CP-91-5402-43, ORA Canada, Ottawa, Ontario, February, 1993.

[L] B.W. Lampson, "A Note on the Confinement Problem," *Comm. ACM, Vol. 16,* No. 10, pp. 613-615, 1973.

[MK] I.S. Moskowitz and M.H. Kang, "Covert Channels --- Here to Stay?," *Proc. COMPASS'94*, 1994.

[MM92] I.S. Moskowitz and A.R. Miller, "The Channel Capacity of a Certain Noisy Timing Channels," *IEEE Trans. Information Theory, Vol. 38*, no.4, pp. 1339-1344, July 1992.

[MM94] I.S. Moskowitz and A.R. Miller, "Simple Timing Channels," *Proc. 1994 IEEE Computer Soc. Symp on Research in Security and Privacy* pp. 56-64, Oakland, Ca., 1994.

[MP] A.P. Moore and C.N. Payne, "Increasing Assurance with Literate Programming Techniques," , " *Proc. 1996 Computer Assurance Conference (COMPASS '96),,* pp. 187-198, Gaithersburg, Md., 1996.

[PM] C.N. Payne and A.P. Moore, "An Experience Modeling Critical Requirements," *Proc. 1994 Computer Assurance Conference (COMPASS '94),* pp. 245-255, Gaithersburg, Md., 1991.

[R] J. Rumbaugh, *et. el. Object Oriented Modeling and Design*, Prentice Hall (1991).

[RST] Reliable Software Technologies, "Whitebox DeepCover: User Reference Manual," RST Corporation, Suite 250, 21515 Ridgetop Circle, Sterling, VA 20166, 1996.

[T] *DoD Trusted Computer System Evaluation Criteria.* DoD 5200.28-STD. 1985.

[S] C. Shannon and W. Weaver, "The Mathematical Theory of Communication," University of Illinois Press (1949).

[VM] J. Voas and K. Miller, "Software Testability: The New Verification. *IEEE Software, 12*(3):17-28, May 1995.

[W] J.C. Wray, "An Analysis of Covert Timing Channels," *Proc. 1991 IEEE Computer Soc. Symp on Research in Security and Privacy* pp. 2-7, Oakland, Ca., 1991.